MULTIMEDIA UNIVERSITY

STUDENT ID NO

MULTIMEDIA UNIVERSITY

FINAL EXAMINATION

TRIMESTER 2, 2015/2016 SESSION

ECE3206 – OBJECT ORIENTED PROGRAMMING WITH C++

(CE, EE, ME & TE)

2 MARCH 2016
9:00 A.M – 11:00 A.M.
(2 Hours)

INSTRUCTIONS TO STUDENT

1. This question paper consists of eight (8) pages only (including this page).
2. **This is an open-book exam**, so the questions are more in-depth and there are fewer questions than normal exams.
3. There are **THREE (3) QUESTIONS** in this paper. Answer **ALL QUESTIONS**. Questions 1 and 2 carry 35 marks each and question 3 carries 30 marks.
4. All programming questions must be answered using the C++ language.
5. Write your answers in the Answer Booklet provided.
6. State all assumptions clearly.

Question 1

The inverse of a square matrix A, sometimes called a reciprocal matrix, is a matrix A^{-1} such that

$$AA^{-1} = A^{-1}A = I \qquad \text{Eq. (1.1)}$$

where I is an identity matrix. For a 2×2 matrix A of $\begin{bmatrix} a & b \\ c & d \end{bmatrix}$, the inverse matrix is:

$$A^{-1} = \frac{1}{\det(A)} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix} \qquad \text{Eq. (1.2)}$$

where $\det(a)$ is the determinant of matrix A and is computed as:

$$\det(A) = ad - bc \qquad \text{Eq. (1.3)}$$

However, if the determinant of a matrix A is 0 (i.e., $\det(a) = 0$), then A does not have an inverse.

Figure Q1-1 illustrates a Unified Modeling Language (UML) class diagram for class **`Matrix`**. This class computes the determinant and the inverse of a 2×2 matrix.

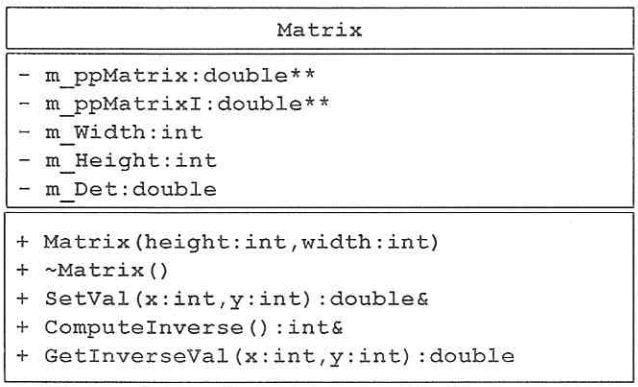

Figure Q1-1: Class `Matrix` UML diagram.

Figure Q1-2 displays the content of a sample 2×2 matrix in a text file.

```
1.80  2.10
5.97  6.18
```

Figure Q1-2: Content of *matrix.txt*.

Using C++ programming language and with reference to Figures Q1-1 and Q1-2:

(a) Implement all the member functions, constructor and destructor of class **`Matrix`**.

i. The **`Matrix()`** constructor saves the input **`height`** and **`width`** parameters, and initializes a set of **two (2)** two-dimension heap (or dynamic) arrays based on the values of these arguments. Use the **`m_ppMatrix`** and **`m_ppMatrixI`** member variables to store the starting addresses of the heap arrays.

[4 marks]

ii. The **`~Matrix()`** destructor frees the heap arrays.

[3 marks]

Continued...

iii. The `SetVal()` member function takes in the `x` (i.e., row) and `y` (i.e., column) coordinate as input arguments. This function returns a reference of `m_ppMatrix` based on the input `x` and `y` coordinate. This means that a reference to the variable in the `m_ppMatrix` heap array (based on `x` and `y` coordinate) is passed back to the caller. This function throws a string exception if the `x` and `y` coordinate exceed the array boundary.

[3 marks]

iv. The `ComputeInverse()` member function first computes the matrix determinant using Eq. (1.3). This function throws a string exception if the computed determinant is zero. Then, using Eq. (1.2), this function computes the matrix inverse into `m_ppMatrixI`.

[7 marks]

v. The `GetInverseVal()` member function takes in the `x` (i.e., row) and `y` (i.e., column) coordinate as input arguments. This function returns the value of `m_ppMatrixI` based on the input `x` and `y` coordinate. This function throws a string exception if the matrix inverse has not been computed. This function throws another string exception if the `x` and `y` coordinate exceed the array boundary.

[4 marks]

(b) Write a `main()` function to test the implemented class `Matrix`.

i. Open the *matrix.txt* text file.

[1 mark]

ii. Instantiate an object of type class `Matrix` using `height = 2` and `width = 2` as constructor arguments.

[2 marks]

iii. Iteratively read the matrix content from the text file. For each value read, set this value into the instantiated `Matrix` object. Call the `SetVal()` member function for this process. As the caller to the `SetVal()` function, use the returned reference to store the read value.

[3 marks]

iv. Call the `ComputeInverse()` member function to compute the inverse of the matrix.

[2 marks]

v. Open a new text file for writing. Iteratively write the content of the inverse matrix into this text file. Call the `GetInverseVal()` member function for this process.

[4 marks]

vi. Close the text files and exit the program.

[2 mark]

Note: *Use C++ file processing when reading from the text file. Apply the necessary* `try` *and* `catch` *procedures in the* `main()` *function.*

Continued...

Question 2

(a) A complex number is a number consisting of a real and imaginary part. It can be described in a form of $x_{real} + x_{img}i$ with i as the standard imaginary unit. The following code excerpt shows a base class called **ComplexNum** that has been defined and the **main()** function for the program. You **do not** need to copy the following code excerpt into your answer.

```
class ComplexNum{
      float m_Real; float m_Img;
public:
      ComplexNum(float inReal = 0.0, float inImg = 0.0){
            m_Real = inReal;
            m_Img = inImg;
      }
};
int main(){
      ComplexNum obj1((float)6.9, (float)8.2);
      ComplexNum obj2((float)25.33, (float)90.82);

      cout<< "Complex Num 1: " << obj1 << endl;
      cout<< "Complex Num 2: " << obj2 << endl;

      ComplexNum obj3 = obj1 / obj2;
      cout << endl << "Complex Num 1 / Complex Num 2 = ";
      cout << obj3 << endl;
      return 0;
}
```

The public constructor of class **ComplexNum** class saves the input of real and imaginary values into class member variables. The driver program divides **obj1** and **obj2** such that Figure Q2-1 below illustrates a sample execution of this program.

```
Complex Num 1: 6.9 + 8.2i
Complex Num 2: 25.33 + 90.82i

Complex Num 1 / Complex Num 2 = 0.103432 + -0.0471268i
```

Figure Q2-1: Sample program execution.

Using the C++ programming language:

i. **Overload the multiplication** (/) operator as a member function to this class. Complex number division is performed algebraically such that:

$$\frac{(x_{real} + x_{img}i)}{(y_{real} + y_{img}i)} = \frac{(x_{real} + x_{img}i)}{(y_{real} + y_{img}i)} \times \frac{(y_{real} - y_{img}i)}{(y_{real} - y_{img}i)}$$

where

$$(x_{real} + x_{img}i) \times (y_{real} - y_{img}i) = (x_{real}y_{real} - x_{img}y_{img}) + (x_{real}y_{img} + x_{img}y_{real})i$$

Continued...

and

$$(y_{real} + y_{img}i) \times (y_{real} - y_{img}i) = (y_{real})^2 + (y_{img})^2$$

[9 marks]

ii. **Overload the stream output operator (<<)** to print out the object values to screen using the format "X_{real} + $X_{img}i$" as shown in Figure Q2-1 above.

[4 marks]

Note: Write parts a (i) - a (ii) as a complete C++ class. As the driver program is already provided, there is ***no need*** *to write one.*

(b) A polynomial linear function has the form:

$$y = ax + b$$

and a univariate (single-variable) polynomial quadratic function has the form:

Note that x represents the single-variable and a, and b are the coefficients. y represents the output of a linear equation for a given x value.

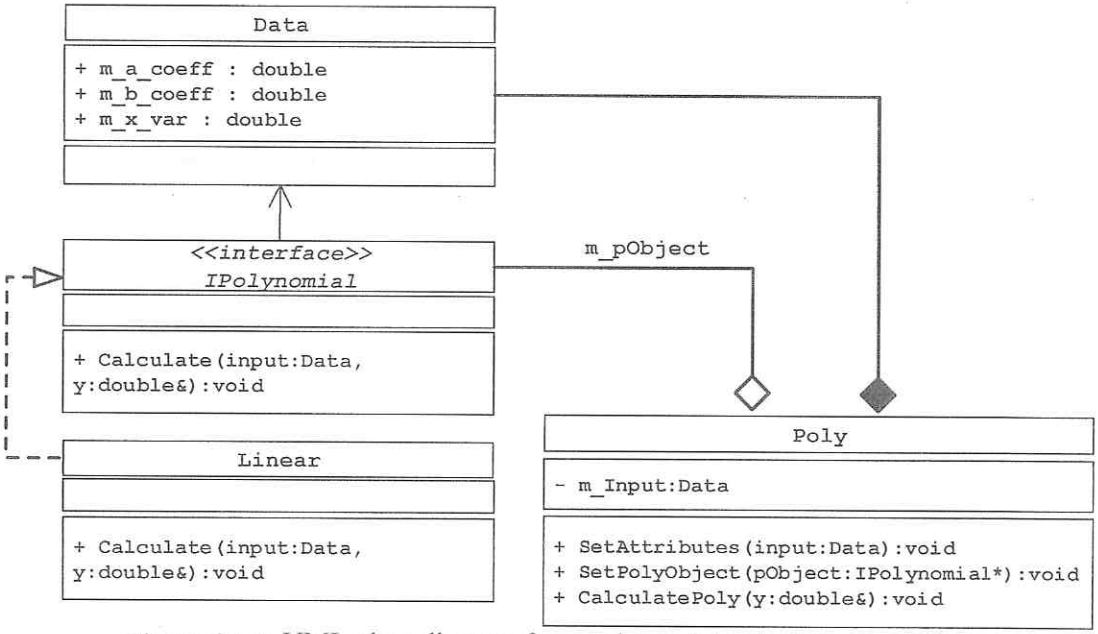

Figure Q2-2: UML class diagram for a polynomial calculation program.

Figure Q2-2 illustrates a **Unified Modeling Language (UML)** class diagram, which describes the design of a polynomial calculation program.

Class **`Data`** stores the values of the a and b coefficients and the x variable. These attributes are used to calculate the output y for a linear function.

The function **`Calculate()`** of class **`Linear`** computes and returns **`y`** as an output argument. Class **`Poly`** sets the input values into the member variables and uses the **`IPolynomial`** class to compute the corresponding polynomial operation. Note that class **`IPolynomial`** is a pure virtual class.

(a) Write complete C++ code, which includes complete class member variables and function definitions for the following classes:

i. **`class Data`**

[1 mark]

Continued...

ii. `class IPolynomial`

[2 marks]

iii. `class Linear`

[3 marks]

iv. `class Poly`

[7 marks]

Refer to the class diagram in Figure Q2-1 to assist in identifying the base and implementation classes.

(b) From part (b), write a `main()` function that will:

i. Instantiate objects of type class `Data` and class `Poly`.

[2 marks]

ii. Prompt the user for the values of the a and b coefficients, x variable, and store these values into the instantiated class `Data` object.

[1 mark]

iii. Set the `Data` object into the instantiated `Poly` object.

[1 mark]

iv. Instantiate an `IPolynomial` object of type class `Linear` and set this object into the instantiated `Poly` object.

[2 marks]

v. Print the results obtained from calling the `CalculatePoly()` method in the instantiated `Poly` object. Apply heap memory clean up at the end of the `main()` function.

[3 marks]

Continued...

Question 3

Figure Q3-1 illustrates a Unified Modeling Language (UML) class diagram for classes **XVar**, **Trigo** and **Sine**. These classes compute the output sine for a given set of x radian values. All of these classes apply a **class template** design.

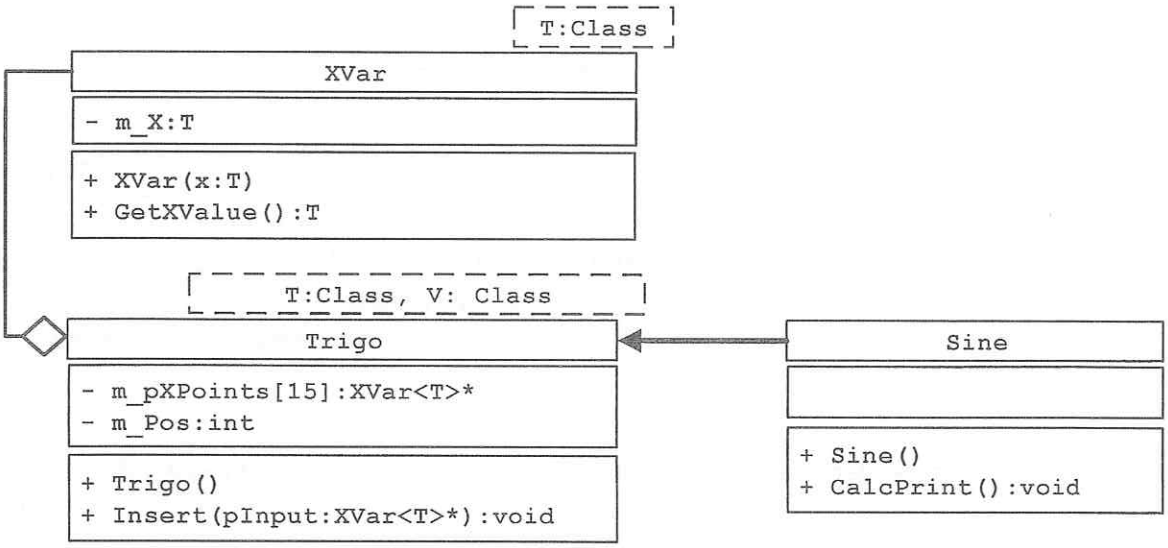

Figure Q3-1: Classes XVar, Trigo, and Sine UML diagram.

(a) Class **XVar** stores the value of the x variable. This class template accepts one template type specification, which can be defined as **template <class T>**. This allows **XVar** to carry out operations for different data types.

Write complete C++ code for class **XVar** with the following details:

i. Proper member variable declaration based on Figure Q3-1.

[2 marks]

ii. The **XVar()** constructor sets the input constructor argument to the class member variable.

[2 marks]

iii. The **GetXValue()** method returns the value of the **T** member variable.

[3 marks]

(b) Class **Trigo** stores an array of fifteen (15) **XVar** type class objects. The **Quadratic** class template accepts three template type specifications, which can be defined as **template <class T, class V>**.

Write complete C++ code for class **Trigo** with the following details:

i. Proper member variable declarations based on Figure Q3-1.

[4 marks]

ii. The **Trigo()** constructor initializes the **m_Pos** member variable to zero. Note that **m_Pos** keeps track of the number items stored into **m_pXPoints**.

[3 marks]

iii. The **Insert()** method inserts the function argument into the **m_pXPoints** array and increments **m_Pos**. This function throws an exception if the array is full.

[4 marks]

(c) Class **Sine** inherits from class **Trigo**. Write complete C++ code for class **Trigo** with the following details:

i. The **CalcPrint()** method computes and prints the sine value for each x value in **m_pXPoints**. Note that the calculate sine value takes the typename **V** placeholder as its data type. Use the **sin()** function from the math library.

[4 marks]

(d) Write a **main()** function to test the **XVar** and **Quadratic** classes.

i. Instantiate an object of type class **Sine** with **<float, double>** type.

[3 marks]

ii. Prompt the user for an x value. Instantiate a heap memory object of type class **XVar** with a **<double>** type. Assign the input x value into this object. (***Hint***: Use C++ dynamic memory allocation to create heap memory). Insert the instantiated **XVar** object into the **Sine** object. Apply the necessary **try** and **catch** procedures here.

[3 marks]

iii. Repeat step (ii) as a loop until user enters a character as an exit value.

[1 mark]

iv. Compute and print out for each input x value using the **CalcPrint()** member function. Refer to Figure Q3-2 to assist you in developing the **main()** function.

[1 mark]

Note: *Apply the necessary* **try** *and* **catch** *procedures in the* **main()** *function. There is no need for heap memory clean up.*

```
Enter a x- radian value (or a char to exit): 4.5

Enter another x- radian value (or a char to exit): 8.6

Enter another x- radian value (or a char to exit): 7.7

Enter another x- radian value (or a char to exit): a

Sine values:

Given x = 4.5, Sin(4.5) = -0.97753
Given x = 8.6, Sin(8.6) = 0.734397
Given x = 7.7, Sin(7.7) = 0.988168
```

Figure Q3-2: Sample program execution.

End of Paper